APPENDIX C
BSDL LISTINGS

Below is a listing of a boundary scan description language (BSDL) file for the IntelDX4 processor.

This file is provided as an example. Contact Intel for design information for this and other Military Intel486 processors. See section 11.5, "Military Intel486 Processor Boundary Scan," for a complete description of BSDL instructions and usage.

IntelDX4 Processor Listing

```
-- Copyright Intel Corporation 1993
--*************************************************************************
-- Intel Corporation makes no warranty for the use of its products
-- and assumes no responsibility for any errors which may appear in
-- this document nor does it make a commitment to update the information
-- contained herein.
--*************************************************************************
-- Boundary-Scan Description Language (BSDL Version 0.0) is a de-facto
-- standard means of describing essential features of ANSI/IEEE 1149.1-1990
-- compliant devices.  This language is under consideration by the IEEE for
-- formal inclusion within a supplement to the 1149.1-1990 standard.  The
-- generation of the supplement entails an extensive IEEE review and a formal
-- acceptance balloting procedure which may change the resultant form of the
-- language.  Be aware that this process may extend well into 1993, and at
-- this time the IEEE does not endorse or hold an opinion on the language.
--*************************************************************************
--
-- IntelDX4(tm) processor BSDL description
-- This file has been electrically verified.
-- ---------------------------------------------------------
-- Rev: 1.2 09/27/93  =


entity IntelDX4 is

   generic (PHYSICAL_PIN_MAP : string := "PGA_17x17");

   port (A20M            :  in    bit;
         ABUS2           :  out   bit;
         ABUS3           :  out   bit;
         ABUS            :  inout bit_vector (4 to 31);  -- Address bus (words)
         ADS             :  out   bit;
         AHOLD           :  in    bit;
         BE              :  out   bit_vector (0 to 3);
         BLAST           :  out   bit;
         BOFF            :  in    bit;
         BRDY            :  in    bit;
         BREQ            :  out   bit;
         BS8             :  in    bit;
         BS16            :  in    bit;
         CLK             :  in    bit;
```

271329-G2

```
        CLKMUL          :  in    bit;
        DBUS            :  inout bit_vector (0 to 31);  -- Data bus
        DC              :  out   bit;
        DP              :  inout bit_vector (0 to 3);
        EADS            :  in    bit;
        FERR            :  out   bit;
        FLUSH           :  in    bit;
        HLDA            :  out   bit;
        HOLD            :  in    bit;
        IGNNE           :  in    bit;
        INC_PGA         :  linkage bit_vector (1 to 5);  -- Internal NC PGA
        INTR            :  in    bit;
        KEN             :  in    bit;
        LOCK            :  out   bit;
        MIO             :  out   bit;
        NC_PGA          :  linkage bit;  -- No Connect for PGA
        NC_SQFP         :  linkage bit_vector (1 to 7);  -- NC SQFP
        NMI             :  in    bit;
        PCD             :  out   bit;
        PCHK            :  out   bit;
        PLOCK           :  out   bit;
        PWT             :  out   bit;
        RDY             :  in    bit;
        RESET           :  in    bit;
        SMI             :  in    bit;
        SMIACT          :  out   bit;
        SRESET          :  in    bit;
        STPCLK          :  in    bit;
        TCK, TMS, TDI   :  in    bit;    -- Scan Port inputs
        TDO             :  out   bit;    -- Scan Port output
        UP              :  in    bit;
        VCC_PGA         :  linkage bit_vector (1 to 23);  -- VCC
        VCC_SQFP        :  linkage bit_vector (1 to 53);  -- VCC
        VCC5            :  linkage bit;  -- Reference Voltage
        VOLDET          :  linkage bit;  -- Voltage Detect Pin, PGA only
        VSS_PGA         :  linkage bit_vector (1 to 28);  -- VSS
        VSS_SQFP        :  linkage bit_vector (1 to 38);  -- VSS
        WR              :  out   bit);

use STD_1149_1_1990.all;

attribute PIN_MAP of IntelDX4 : entity is PHYSICAL_PIN_MAP;

constant PGA_17x17 : PIN_MAP_STRING :=            -- Define Pin Out of PGA

     "A20M    : D15, "&
     "ABUS2   : Q14, "&
     "ABUS3   : R15, "&
     "ABUS    : (S16,Q12,S15,Q13,R13,Q11,S13,R12,"&
     "            S07,Q10,S05,R07,Q09,Q03,R05,Q04,Q08,Q05,"&
     "            Q07,S03,Q06,R02,S02,S01,R01,P02,P03,Q01),"&
     "ADS     : S17, "&
     "AHOLD   : A17, "&
     "BE      : (K15,J16,J15,F17),"&
     "BLAST   : R16, "&
     "BOFF    : D17, "&
```

271329-G3

```
      "BRDY     : H15, "&
      "BREQ     : Q15, "&
      "BS8      : D16, "&
      "BS16     : C17, "&
      "CLK      : C03, "&
      "CLKMUL   : R17, "&
      "DBUS     : (P01,N02,N01,H02,M03,J02,L02,L03,F02,D01,E03,"&
      "             C01,G03,D02,K03,F03,J03,D03,C02,B01,A01,B02,"&
      "             A02,A04,A06,B06,C07,C06,C08,A08,C09,B08)," &
      "DC       : M15, "&
      "DP       : (N03,F01,H03,A05),"&
      "EADS     : B17, "&
      "FERR     : C14, "&
      "FLUSH    : C15, "&
      "HLDA     : P15, "&
      "HOLD     : E15, "&
      "IGNNE    : A15, "&
      "INC_PGA  : (A10,A12,A13,B12,B13),"&
      "INTR     : A16, "&
      "KEN      : F15, "&
      "LOCK     : N15, "&
      "MIO      : N16, "&
      "NC_PGA   : C13, "&
      "NMI      : B15, "&
      "PCD      : J17, "&
      "PCHK     : Q17, "&
      "PLOCK    : Q16, "&
      "PWT      : L15, "&
      "RDY      : F16, "&
      "RESET    : C16, "&
      "SMI      : B10, "&
      "SMIACT   : C12, "&
      "SRESET   : C10, "&
      "STPCLK   : G15, "&
      "TCK      : A03, "&
      "TDI      : A14, "&
      "TDO      : B16, "&
      "TMS      : B14, "&
      "UP       : C11, "&
      "VCC_PGA  : (B07,B09,B11,C04,C05,E2,E16,G02,G16,H16,K02,"&
      "             K16,L16,M02,M16,P16,R03,R06,R08,R09,R10,R11,"&
      "             R14),"&
      "VCC5     : J01, "&
      "VOLDET   : S04, "&
      "VSS_PGA  : (A07,A09,A11,B03,B04,B05,E01,E17,G01,G17,H01,H17,"&
      "             K01,K17,L01,L17,M01,M17,P17,Q02,R04,S06,S08,S09,"&
      "             S10,S11,S12,S14),"&
      "WR       : N17  ";


constant SQFP_208 : PIN_MAP_STRING :=          -- Define Pin Out of SQFP

      "A20M     : 47,  "&
      "ABUS2    : 202, "&
      "ABUS3    : 197, "&
```

271329-G4

```
"ABUS     : (196,195,193,192,190,187,186,182,180,178,"&
"            177,174,173,171,166,165,164,161,160,159,"&
"            158,154,153,152,151,149,148,147),"  &
"ADS      : 203, "&
"AHOLD    : 17,  "&
"BE       : (31,32,33,34),"&
"BLAST    : 204, "&
"BOFF     : 6,   "&
"BRDY     : 5,   "&
"BREQ     : 30,  "&
"BS8      : 8,   "&
"BS16     : 7,   "&
"CLK      : 24,  "&
"CLKMUL   : 11,  "&
"DBUS     : (144,143,142,141,140,130,129,126,124,123,119,"&
"            118,117,116,113,112,108,103,101,100,99,93,"&
"            92,91,87,85,84,83,79,78,75,74),"&
"DC       : 39,  "&
"DP       : (145,125,109,90),"&
"EADS     : 46,  "&
"FERR     : 66,  "&
"FLUSH    : 49,  "&
"HLDA     : 26,  "&
"HOLD     : 16,  "&
"IGNNE    : 72,  "&
"INTR     : 50,  "&
"KEN      : 13,  "&
"LOCK     : 207, "&
"MIO      : 37,  "&
"NC_SQFP  : (63,64,67,70,71,96,127),"&
"NMI      : 51,  "&
"PCD      : 41,  "&
"PCHK     : 4,   "&
"PLOCK    : 206, "&
"PWT      : 40,  "&
"RDY      : 12,  "&
"RESET    : 48,  "&
"SMI      : 65,  "&
"SMIACT   : 59,  "&
"SRESET   : 58,  "&
"STPCLK   : 73,  "&
"TCK      : 18,  "&
"TDI      : 168, "&
"TDO      : 68,  "&
"TMS      : 167, "&
"UP       : 194, "&
"VCC_SQFP : (2,9,14,19,20,22,23,25,29,35,38,42,44,45,54,"&
"            56,60,62,69,77,80,82,86,89,95,98,102,106,111,"&
"            114,121,128,131,133,134,136,137,139,150,155,"&
"            162,163,169,172,176,179,183,185,188,191,198,"&
"            200,205),"&
"VCC5     : 3,   "&
"VSS_SQFP : (1,10,15,21,28,36,43,52,53,55,57,61,76,81,88,94,"&
"            97,104,105,107,110,115,120,122,132,135,138,146,"&
"            156,157,170,175,181,184,189,199,201,208),"&
"WR       : 27 ";
```

271329-G5

```
   attribute Tap_Scan_In of    TDI : signal is true;
   attribute Tap_Scan_Out of   TDO : signal is true;
   attribute Tap_Scan_Mode of  TMS : signal is true;

   attribute Tap_Scan_Clock of TCK : signal is (25.0e6, BOTH);

   attribute Instruction_Length of IntelDX4 : entity is 4;

   attribute Instruction_Opcode of IntelDX4 : entity is

      "BYPASS   (1111)," &
      "EXTEST   (0000)," &
      "SAMPLE   (0001)," &
      "IDCODE   (0010)," &
      "RUNBIST  (1000)," &
      "PRIVATE  (0011,0100,0101,0110,0111,1001,1010,1011,1100,1101,1110)";

   attribute Instruction_Capture of IntelDX4 : entity is "0001";

   -- there is no Instruction_Disable attribute for IntelDX4

   attribute Instruction_Private of IntelDX4 : entity is "private";

   attribute Idcode_Register of IntelDX4: entity is
-- ******************************************
      "0000"                    &  --version
      "1000001010001000"&  --new part number
      "00000001001"     &  --manufacturers identity
      "1";                          --required by the standard

   attribute Instruction_Usage of IntelDX4 : entity is
      "RUNBIST (registers BIST; "&
      "result 0;"          &
      "clock CLK in Run_Test_Idle;"&
      "length 1600000)";

   attribute Register_Access of IntelDX4 : entity is
      "BIST[1] (RUNBIST)";

--{*********************************************************************}
--{  The first cell is closest to TDO                                    }
--{*********************************************************************}

   attribute Boundary_Length of IntelDX4 : entity is 109;
   attribute Boundary_Cells of IntelDX4 : entity is "BC_2, BC_1, BC_6";

   attribute Boundary_Register of IntelDX4 : entity is
      "0        (BC_2,  ABUS2,          output3,        X,  107,  1,  Z),"&
      "1        (BC_2,  ABUS3,          output3,        X,  107,  1,  Z),"&
      "2        (BC_6,  ABUS(4),        bidir,          X,  107,  1,  Z),"&
      "3        (BC_6,  ABUS(5),        bidir,          X,  107,  1,  Z),"&
      "4        (BC_1,  UP,             input,          X),"         &
      "5        (BC_6,  ABUS(6),        bidir,          X,  107,  1,  Z),"&
      "6        (BC_6,  ABUS(7),        bidir,          X,  107,  1,  Z),"&
      "7        (BC_6,  ABUS(8),        bidir,          X,  107,  1,  Z),"&
```

271329-G6

```
"8        (BC_6,  ABUS(9),        bidir,          X,  107,  1,  Z),"&
"9        (BC_6,  ABUS(10),       bidir,          X,  107,  1,  Z),"&
"10       (BC_6,  ABUS(11),       bidir,          X,  107,  1,  Z),"&
"11       (BC_6,  ABUS(12),       bidir,          X,  107,  1,  Z),"&
"12       (BC_6,  ABUS(13),       bidir,          X,  107,  1,  Z),"&
"13       (BC_6,  ABUS(14),       bidir,          X,  107,  1,  Z),"&
"14       (BC_6,  ABUS(15),       bidir,          X,  107,  1,  Z),"&
"15       (BC_6,  ABUS(16),       bidir,          X,  107,  1,  Z),"&
"16       (BC_6,  ABUS(17),       bidir,          X,  107,  1,  Z),"&
"17       (BC_6,  ABUS(18),       bidir,          X,  107,  1,  Z),"&
"18       (BC_6,  ABUS(19),       bidir,          X,  107,  1,  Z),"&
"19       (BC_6,  ABUS(20),       bidir,          X,  107,  1,  Z),"&
"20       (BC_6,  ABUS(21),       bidir,          X,  107,  1,  Z),"&
"21       (BC_6,  ABUS(22),       bidir,          X,  107,  1,  Z),"&
"22       (BC_6,  ABUS(23),       bidir,          X,  107,  1,  Z),"&
"23       (BC_6,  ABUS(24),       bidir,          X,  107,  1,  Z),"&
"24       (BC_6,  ABUS(25),       bidir,          X,  107,  1,  Z),"&
"25       (BC_6,  ABUS(26),       bidir,          X,  107,  1,  Z),"&
"26       (BC_6,  ABUS(27),       bidir,          X,  107,  1,  Z),"&
"27       (BC_6,  ABUS(28),       bidir,          X,  107,  1,  Z),"&
"28       (BC_6,  ABUS(29),       bidir,          X,  107,  1,  Z),"&
"29       (BC_6,  ABUS(30),       bidir,          X,  107,  1,  Z),"&
"30       (BC_6,  ABUS(31),       bidir,          X,  107,  1,  Z),"&
"31       (BC_6,  DP(0),          bidir,          X,  108,  1,  Z),"&
"32       (BC_6,  DBUS(0),        bidir,          X,  108,  1,  Z),"&
"33       (BC_6,  DBUS(1),        bidir,          X,  108,  1,  Z),"&
"34       (BC_6,  DBUS(2),        bidir,          X,  108,  1,  Z),"&
"35       (BC_6,  DBUS(3),        bidir,          X,  108,  1,  Z),"&
"36       (BC_6,  DBUS(4),        bidir,          X,  108,  1,  Z),"&
"37       (BC_6,  DBUS(5),        bidir,          X,  108,  1,  Z),"&
"38       (BC_6,  DBUS(6),        bidir,          X,  108,  1,  Z),"&
"39       (BC_6,  DBUS(7),        bidir,          X,  108,  1,  Z),"&
"40       (BC_6,  DP(1),          bidir,          X,  108,  1,  Z),"&
"41       (BC_6,  DBUS(8),        bidir,          X,  108,  1,  Z),"&
"42       (BC_6,  DBUS(9),        bidir,          X,  108,  1,  Z),"&
"43       (BC_6,  DBUS(10),       bidir,          X,  108,  1,  Z),"&
"44       (BC_6,  DBUS(11),       bidir,          X,  108,  1,  Z),"&
"45       (BC_6,  DBUS(12),       bidir,          X,  108,  1,  Z),"&
"46       (BC_6,  DBUS(13),       bidir,          X,  108,  1,  Z),"&
"47       (BC_6,  DBUS(14),       bidir,          X,  108,  1,  Z),"&
"48       (BC_6,  DBUS(15),       bidir,          X,  108,  1,  Z),"&
"49       (BC_6,  DP(2),          bidir,          X,  108,  1,  Z),"&
"50       (BC_6,  DBUS(16),       bidir,          X,  108,  1,  Z),"&
"51       (BC_6,  DBUS(17),       bidir,          X,  108,  1,  Z),"&
"52       (BC_6,  DBUS(18),       bidir,          X,  108,  1,  Z),"&
"53       (BC_6,  DBUS(19),       bidir,          X,  108,  1,  Z),"&
"54       (BC_6,  DBUS(20),       bidir,          X,  108,  1,  Z),"&
"55       (BC_6,  DBUS(21),       bidir,          X,  108,  1,  Z),"&
"56       (BC_6,  DBUS(22),       bidir,          X,  108,  1,  Z),"&
"57       (BC_6,  DBUS(23),       bidir,          X,  108,  1,  Z),"&
"58       (BC_6,  DP(3),          bidir,          X,  108,  1,  Z),"&
"59       (BC_6,  DBUS(24),       bidir,          X,  108,  1,  Z),"&
"60       (BC_6,  DBUS(25),       bidir,          X,  108,  1,  Z),"&
"61       (BC_6,  DBUS(26),       bidir,          X,  108,  1,  Z),"&
"62       (BC_6,  DBUS(27),       bidir,          X,  108,  1,  Z),"&
"63       (BC_6,  DBUS(28),       bidir,          X,  108,  1,  Z),"&
```

271329-G7

```
      "64       (BC_6,  DBUS(29),       bidir,          X,  108,  1,  Z),"&
      "65       (BC_6,  DBUS(30),       bidir,          X,  108,  1,  Z),"&
      "66       (BC_6,  DBUS(31),       bidir,          X,  108,  1,  Z),"&
      "67       (BC_2,  STPCLK,         input,          X),"&
      "68       (BC_1,  IGNNE,          input,          X),"&
      "69       (BC_2,  FERR,           output3,        X,  105,  1,  Z),"&
      "70       (BC_1,  SMI,            input,          X),"&
      "71       (BC_2,  SMIACT,         output3,        X,  106,  1,  Z),"&
      "72       (BC_1,  SRESET,         input,          X),"&
      "73       (BC_1,  NMI,            input,          X),"&
      "74       (BC_1,  INTR,           input,          X),"&
      "75       (BC_1,  FLUSH,          input,          X),"&
      "76       (BC_1,  RESET,          input,          X),"&
      "77       (BC_1,  A20M,           input,          X),"&
      "78       (BC_1,  EADS,           input,          X),"&
      "79       (BC_2,  PCD,            output3,        X,  106,  1,  Z),"&
      "80       (BC_2,  PWT,            output3,        X,  106,  1,  Z),"&
      "81       (BC_2,  DC,             output3,        X,  106,  1,  Z),"&
      "82       (BC_2,  MIO,            output3,        X,  106,  1,  Z),"&
      "83       (BC_2,  BE(3),          output3,        X,  106,  1,  Z),"&
      "84       (BC_2,  BE(2),          output3,        X,  106,  1,  Z),"&
      "85       (BC_2,  BE(1),          output3,        X,  106,  1,  Z),"&
      "86       (BC_2,  BE(0),          output3,        X,  106,  1,  Z),"&
      "87       (BC_2,  BREQ,           output3,        X,  105,  1,  Z),"&
      "88       (BC_2,  WR,             output3,        X,  106,  1,  Z),"&
      "89       (BC_2,  HLDA,           output3,        X,  105,  1,  Z),"&
      "90       (BC_1,  CLK,            input,          X),"&
      "91       (BC_1,  AHOLD,          input,          X),"&
      "92       (BC_1,  HOLD,           input,          X),"&
      "93       (BC_1,  KEN,            input,          X),"&
      "94       (BC_1,  RDY,            input,          X),"&
      "95       (BC_1,  CLKMUL,         input,          X),"&
      "96       (BC_1,  BS8,            input,          X),"&
      "97       (BC_1,  BS16,           input,          X),"&
      "98       (BC_1,  BOFF,           input,          X),"&
      "99       (BC_1,  BRDY,           input,          X),"&
      "100      (BC_2,  PCHK,           output3,        X,  105,  1,  Z),"&
      "101      (BC_2,  LOCK,           output3,        X,  106,  1,  Z),"&
      "102      (BC_2,  PLOCK,          output3,        X,  106,  1,  Z),"&
      "103      (BC_2,  BLAST,          output3,        X,  106,  1,  Z),"&
      "104      (BC_2,  ADS,            output3,        X,  106,  1,  Z),"&
      "105      (BC_2,  *,      control, 1),"&              -- DISMISC
      "106      (BC_2,  *,      control, 1),"&              -- DISBUS
      "107      (BC_2,  *,      control, 1),"&              -- DISABUS
      "108      (BC_2,  *,      control, 1)";               -- DISWR

end IntelDX4;
```

271329-G8

APPENDIX D
SYSTEM DESIGN NOTES

SMM Environment Initialization

When the Military Intel486 processors are operating in Real Mode, the physical address at which instructions and data are fetched is determined by the segment register and an offset (i.e., CS and IP for instructions). When a new value is loaded into a segment register, the new value is shifted to the left by four bits and stored in a segment base register that corresponds to that particular segment (CSBASE, DSBASE, ESBASE, etc.). It is the value stored in the segment base register that is actually used to generate a physical address. For example, the linear address to be used for fetching instructions is determined by adding the value contained in the CS segment base register with the value in the IP register.

When the processor is in Protected Mode, the segment registers are used as selectors to a descriptor table. Each descriptor in a descriptor table contains information about the segment in use, including the segments BASE address (i.e., CSBASE), the limit (or size of the segment), as well as protection level, privileges, operand sizes, and the segment type. In Protected Mode, the linear address is determined by adding the base portion of the descriptor to the appropriate offset.

When in System Management Mode, the processor operates in a pseudo-Real Mode, with address calculation performed in the Real Mode manner. However, the processor adds the value in the segment base register with the value in the EIP register, rather than the IP register, so there are no limits as to the segment size. The physical address of an instruction is obtained by adding the value in CSBASE to the value in EIP.

When entering SMM, it may be necessary to initialize the segment registers to point to SMRAM (see section 8.4.2, ‘‘Processor Environment,’’ for their value on SMM entry). If SMBASE has not been relocated, then the necessary segment registers can be initialized to point to SMRAM by using the value in the CS register, 3000H, which points to the SMRAM address space.

When an SMI# occurs after SMBASE has been modified, CSBASE is loaded with the new value of SMBASE. **However, the CS selector register still contains the value 3000H, not the value corresponding to the new SMBASE**.

To initialize segment registers to point to the new SMRAM area, read the SMBASE value from the SMM state that was saved in memory. Because the data segment registers are initialized to 0, do not use them to access the SMM state save area. Instead, perform a read relative to the CS register by using a CS override prefix to a normal memory read. Although CS still contains 3000H, CSBASE contains the value of SMBASE, and CSBASE is used for the address generation.

Once the value of SMBASE is obtained, it must be shifted to the right by four bits to get the appropriate value to be placed in the segment registers. The CS register itself can be initialized by executing a far jump instruction to an address within SMBASE, which causes CS to be reloaded with a value corresponding to SMBASE.

Example D-1 describes one method of initializing the segment registers when SMBASE has been relocated. This method works if SMBASE is less than 1 Megabyte.

Example D-1. Initialization of Segment Registers within SMM

```
;read the value of SMBASE from the state save area
      mov   si,FEF8H     ;SMBASE slot in SMM state save area
      mov   eax,cs:[si]  ;copy SMBASE from SMBASE:FEF8H to eax

;scale the SMBASE value to a 16-bit quantity
      mov   cl,4
      ror   eax,cl       ;scaled value of SMBASE now in ax

;to load cs, execute a far jump to an address that has been stored
;at memory location PTR_ADDR

;store the SMBASE value and an offset to a memory location that can be used as
;an indirect jump address

      mov   di,PTR_ADDR  ;PTR_ADDR is the location used to
                         ;store the jump address
      mov   bx,OFFSET    ;OFFSET is the address where
                         ;execution continues after the
                         ;far jump

      mov   cs:[di],bx   ;store the offset for the far jump
      inc   di
      inc   di
      mov   cs:[di],ax   ;store the segment address for the
                         ;far jump, which is SMBASE
      mov   bx,PTR_ADDR  ;bx now contains the address of the
                         ;location holding the jump address

;initialize DS and ES with the correct address of SMBASE
      mov   ds,ax
      mov   es,ax

;execute a far jump instruction to load the CS register
      jmp   far [bx]     ;jump to address stored at memory
                         ;location pointed to by bx

;CS now contains the correct value of SMBASE, and execution continues from the
;address SMBASE:OFFSET
```

271329–G9

Accessing SMRAM

LOADING SMRAM WITH AN INITIAL SMI HANDLER

Under normal conditions, the SMRAM address space should only be accessible by the processor while it is in SMM mode. However, some provision must be made for providing the initial SMM interrupt handler routine.

Because System Management Mode must be transparent to all operating systems, the initial SMM handler must be loaded by the system BIOS. At some time during the power on sequence, the system BIOS will need to move the SMM handler routine from the BIOS ROM to the SMRAM. The system designer must provide a hardware mechanism that allows access to SMRAM while SMIACT# from the processor is inactive. One method would be to provide an I/O port in the memory controller that forces

memory cycles at a given address to be relocated to the SMRAM. Once the initial SMM handler has been loaded to SMRAM, the I/O port would be disabled to protect against accidental accesses to SMRAM.

The system BIOS must provide an SMM handler at the address 38000H. If the system designer has chosen to take advantage of the SMRAM relocation feature of the processor, this handler must change the SMBASE register in the SMM state save. Next, the BIOS must move the full featured SMM handler to the new address. An SMI# must be generated in order to change the SMBASE register before the BIOS passes control to the operating system.

SMRAM HIDDEN FROM DMA AND BUS MASTERS

In a system that allows DMA or other devices to take control of the system bus, care must be taken to ensure that only the master processor can access SMRAM. If an external bus master requests use of the system bus (by asserting HOLD or BOFF#) while the processor is executing an SMM handler routine, the processor would respond by passing control of the bus to the requesting device. The system memory controller must redirect any memory accesses that are not generated by the processor to normal system memory as if SMIACT# was inactive. DMA accesses to the SMRAM area must be redirected to the correct address space when the initialization routine is loading SMRAM, as well as when the processor is in SMM.

It is not recommended to block bus control requests when in SMM, because the increased bus access latency could cause compatibility issues with some software or expansion hardware.

ACCESSING SYSTEM MEMORY FROM WITHIN SMM

In order to enter a suspend state where power is removed from some or all of system memory, it is necessary for the processor to have access to the entire system address space from within SMM. Access to system memory from within SMM requires that the memory controller decode both SMIACT# and the processor address to determine accesses to SMRAM. Only those memory addresses that are defined as being SMRAM space would be directed to SMRAM. If SMRAM is located at an address that overlays normal system memory address space (see section 8.6.1, ''SMRAM Interface,''.), the processor must have a method of accessing both SMRAM (for code reads) and system memory simultaneously.

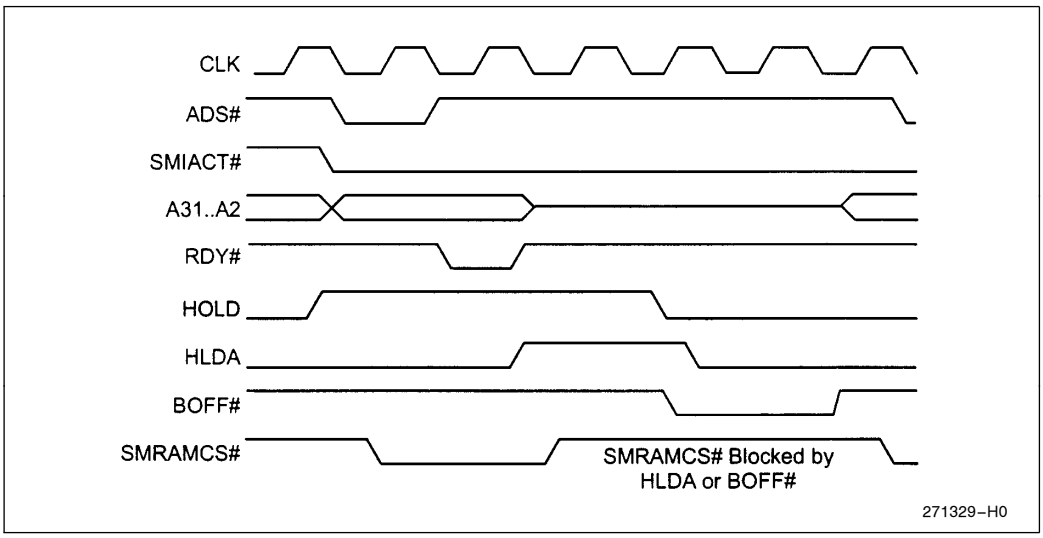

Figure D-1. Blocking Other Bus Masters from Accessing SMRAM

Ideally, a method of accessing system memory that is mapped underneath SMRAM would be provided by the system memory controller. The memory controller would provide a register that allows system memory at a given address to be remapped to a different address, which is not overlaid by SMRAM. When the SMM handler implements a suspend, it would first move all of system memory that is not underneath SMRAM to a non-volatile medium (such as a hard disk drive). Next, the SMRAM image would be transferred to the non-volatile medium. Finally, the memory underneath SMRAM would be accessed and copied to the non-volatile medium with a processor read to the remap address space, which is redirected to the overlaid system memory (see Figure D-2).

If the memory controller does not provide a method of accessing overlaid system memory, it is possible to implement a software procedure to accomplish the same goal. However, the software method is quite complex, and a hardware method is preferred. A description of the software method follows.

The ability to access the system memory that is located in the address space under SMRAM requires a method of resuming from SMM to a predetermined address space. This can be accomplished with the following procedure.

When resuming from SMM, the processor continues execution at the address contained in the CS and EIP slots within the SMM state save. However, the resume address cannot be changed by simply modifying the CS and EIP slots, because the processor will use the CS descriptor to determine the actual resume address. The descriptor registers are stored in reserved slots in the SMM state save, and they cannot be directly modified.

By replacing the suspend state save with a previously obtained image of a state save that returns to a known location, the SMM suspend handler can force a return to a given address:

1. During initial system power up, execute an SMI# from a predetermined address (the address immediately preceding the address to which you later wish to resume). This can be accomplished by generating an SMI# in response to an I/O instruction or executing a halt instruction and using an SMI# to exit the halt state.
2. Save the state save from this SMM to a safe location (SMRAM).
3. When the system needs to resume to a given address from some other SMI#, the stored state save can be substituted for the state save generated from that particular SMM.

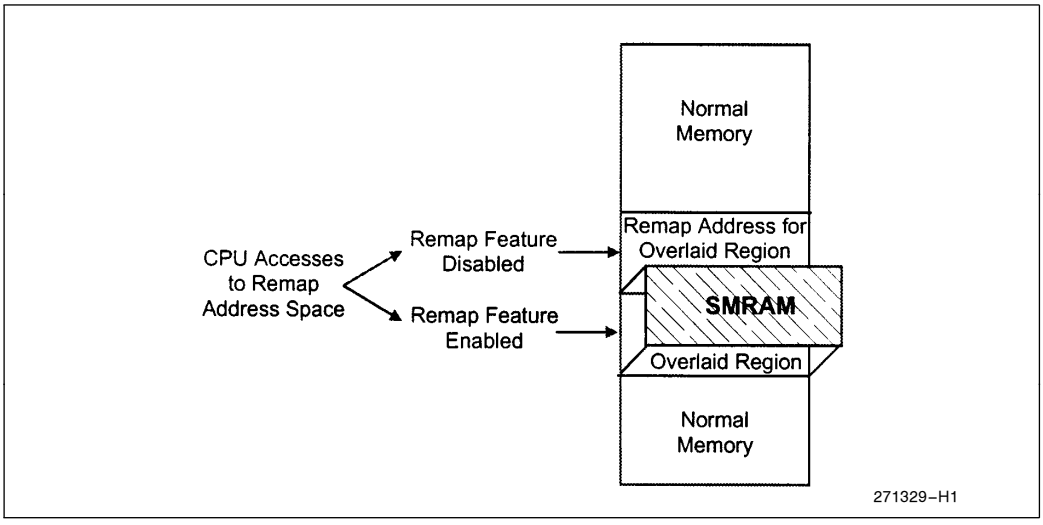

Figure D-2. Remapping Memory That Is Overlaid by SMRAM

Now that SMM can be resumed at a predetermined address, access the entire system memory space from within SMM before executing a suspend:

1. During a suspend SMM, save all system memory except that which is located underneath SMRAM to a specified (and reserved) section of the hard disk. The ability to access system memory requires the memory controller to decode both SMIACT# and the processor address, and direct a limited section (maybe 64 or 128K) of the processor address space to SMRAM. All other processor memory accesses should go to normal system memory.
2. Save the contents of the SMM state save to the hard disk.
3. Modify the SMM state save so that the RSM instruction will return to a predefined address, which is not in the application that was interrupted. The code at this address must contain the remainder of the suspend SMM handler. The predefined address can be anywhere in the processor address space, because the contents of system memory have already been saved to disk.
4. Execute an RSM instruction, which exits SMM and returns control to a predetermined address (which must contain the rest of the SMM suspend handler).
5. Save the rest of system memory (that which is located underneath SMRAM) to the hard disk. This address space can now be accessed with normal move instructions, because we are no longer in SMM.
6. Save a flag (in CMOS memory) indicating that the next reset should cause a resume from suspend.
7. Powerdown the memory (and possibly the processor).
8. When power is restored, the processor is reset and begins execution of the POST in BIOS. Early in the POST, the system should check the status of the suspend flag.
9. Load a preliminary SMM handler to location 38000H and generate an SMI#. The SMM handler should read the SMBASE slot from the SMM state save that was stored to hard disk. SMBASE is then modified to point to the final SMRAM location and the system resumes from SMM back to the system BIOS.
10. Restore the contents of system memory located underneath SMRAM from the hard disk.
11. Generate a second SMI#, which executes an SMM handler at the original value of SMBASE (before the suspend SMM). The SMM handler restores the contents of the rest of system memory from the hard disk, and then restores the original SMM state save to the SMM state save area in SMRAM, discarding the most recent SMM state save.
12. Execute an RSM instruction, which returns execution to the application that was interrupted by the suspend request.

Interrupts and Exceptions During SMM Handler Routines

To ensure transparency to existing system software, the SMM handler should not depend on interrupt or exception handlers provided by the operating system. However, in some cases it may be necessary to service interrupts or exceptions while in System Management Mode. In these cases, SMM compliant interrupt and exception handlers, as well as an SMM compliant interrupt vector table, should be provided.

SMM COMPLIANT VECTOR TABLES

An SMI# interrupt request can be generated while code is running under any of the other three processor operating modes (Real, Virtual-86, or Protected). When entering the SMM handler, the processor enters a pseudo-real mode, and the beginning of the interrupt vector table must be located at the address 00000000H. Before allowing any interrupts or exceptions to occur, the SMM handler routine must provide a valid interrupt vector table. Any code that is executed before setting up an SMM compliant interrupt vector table must be written carefully to ensure that no exceptions are generated.

The system memory controller could relocate accesses to the SMM interrupt vector table to a location within SMRAM. In this case, when SMIACT# is active, all accesses to the lowest 1 Kbyte of the processor address space would be redirected to SMRAM, which would contain an SMM compliant vector table that has already been initialized.

If the system memory controller does not redirect interrupt vector table reads to an address within SMRAM, there are three steps required to provide an SMM compliant interrupt vector table:

1. Save the contents of memory at address 00000000H to SMRAM

2. Provide vectors for any possible interrupts or exceptions at the appropriate location in the vector table
3. Restore the original memory contents from SMRAM before exiting the SMM handler routine

INTERRUPTS AND SUBROUTINES WITH SMRAM RELOCATION

There is an additional issue that must be considered if interrupts or exceptions are to be executed within SMM and SMRAM has been relocated. Interrupt or subroutine calls from within SMM operate in a manner similar to Real Mode. When a subroutine is called or an interrupt is recognized, the 16-bit CS and IP registers are pushed onto the stack to provide a return address.

When SMRAM is relocated to an address space above 1M and an interrupt or subroutine call occurs, only 16 bits of the EIP register are pushed onto the stack. When returning from the subroutine or interrupt, the processor will vector to a location where the upper 16 bits of EIP are zero. This can be avoided for subroutines by using an address size override before calling the subroutine. However, the issue remains for interrupts.

Military Intel486 DX, IntelDX2, and IntelDX4 Processor Floating Point Operation and SMM

THE NEED TO SAVE THE FPU ENVIRONMENT

When the processor enters System Management Mode, the context information for the interrupted application is automatically saved to a specific state save address. When the SMM handler returns control to the interrupted application by executing the RSM instruction, the context information from the interrupted application is restored to the processor by reading from the state save location. This mechanism allows the SMM handler routine to modify most of the processor registers without the need to explicitly save them to memory. However, the registers in the processor's Floating Point Unit (FPU) are not automatically saved when the processor enters SMM. If the SMM handler needs to modify any of the registers in the FPU, or if the register data will be lost due to entering a power down state, the SMM handler must first explicitly save the FPU state as it existed in the interrupted application.

There are two instances in which an SMM handler routine must be aware of the Floating Point Unit (FPU):

1. When removing power from the processor / FPU for the purpose of executing a suspend sequence.
2. When the SMM handler uses FPU instructions.

In both of these cases, the SMM handler must save the state of the FPU as it was left by the interrupted application.

The information stored by the FPU state save instructions (FSAVE, FNSAVE, FSTENV, and FNSTENV) is dependent on the operating mode of the processor. The FPU state save instructions store the FPU state information in one of four formats: 16-bit Real Mode, 32-bit Real Mode, 16-bit Protected Mode, or 32-bit Protected Mode, depending on the processor operating mode. The content of the information saved also varies slightly, depending on the processor operating mode in which the save instruction was executed. For example, the 32-bit Protected Mode FNSAVE instruction saves the address of the last executed FPU instruction and its operands in the form of a segment selector and a 32-bit offset. In contrast, the 16-bit Real Mode FNSAVE instruction saves the address information in the form of a 20 bit physical address. Because the format with which the FPU state restore instructions (FRSTOR and FLDENV) recall the information is also dependent on the operating mode of the processor, the save and restore instructions must be executed from the same processor operating mode.

SAVING THE STATE OF THE FLOATING POINT UNIT

When an SMM handler routine needs to save the state of the Floating Point Unit, it must save all FPU state information necessary for the interrupted application to continue processing. This state information includes the contents of the Floating Point Unit stack, which requires use of the FNSAVE or FSAVE instruction (FSTENV does not save the contents of the FPU stack). If the last executed non-control Floating Point instruction caused an error (such as a divide by 0), the saved information must also include the address of the failing instruction and the addresses of any operands for that instruction. Without these addresses, it would be impossible for the FPU exception handler of the interrupted application to correct the error and restart the instruction.

The FNSAVE and FSAVE instructions differ in that FNSAVE does not wait for the FPU to check for an existing error condition before storing the FPU environment information. If there is an unmasked FPU exception condition pending, execution of the FSAVE instruction will force the processor to wait until the error condition is cleared by the software exception handler. Because the processor is in System Management Mode, the appropriate exception handler will not be available, and the FPU error would not be corrected in the manner expected by the interrupted application program. For this reason, the FNSAVE instruction should be used when saving the environment of the FPU within SMM.

Because the SMM handler does not know the processor mode in which the interrupted application was executing (16 or 32 bit, Real or Protected), the SMM handler must execute the FNSAVE instruction in a mode in which all FPU state information is stored. The 32-bit Protected Mode format of the FNSAVE instruction is a super set of all other formats of the FNSAVE instruction. Therefore, executing the 32-bit Protected Mode FNSAVE instruction ensures that all FPU state information will be saved.

Executing the FNSAVE instruction in 32-bit Protected Mode requires that the processor be temporarily placed in Protected Mode. Rather than perform all of the setup details and overhead necessary to place the processor into Protected Mode, including the initialization of all descriptors and descriptor tables, it is possible to temporarily place the processor into Protected Mode for the purpose of executing only a few carefully written instructions. This can be accomplished by setting the PE bit in the CR0 register, and then executing a short jump to clear the instruction pipelines.

It is important to note that any instruction that modifies a segment register will cause the processor to attempt to load a new descriptor from the descriptor table. (The occurrence of an interrupt or an exception would cause the processor to load a new descriptor, so interrupts must be disabled during this sequence.) Because neither the descriptors nor the descriptor table have been initialized, this would cause the system to crash. Therefore, all segment registers that are to be used in the FPU state save instructions must be initialized before entering Protected Mode.

Example D-2 gives an example of the code that can be used to place the processor in Protected Mode and save the FPU state.

Note that the no wait form (FNSAVE) of the save instruction must be used. In the event that the previous FPU instruction caused a floating point error, we do not want to wait for this error to be serviced before executing the save instruction. Additionally, if the FSAVE instruction were used, the operand size override prefix would be incorrectly applied to the implicit WAIT instruction which precedes FSAVE, rather than to the save instruction itself.

Before exiting the SMM handler and returning to the interrupted application, the register contents of the Floating Point Unit must be returned to their previous values. This can be accomplished by executing the 32-bit Protected Mode format of the FRSTOR instruction. Example D-3 gives an example code segment that can be used to restore the FPU to the state in which it was interrupted by the SMI request.

Note that the no wait form (FNRSTOR) of the restore instruction must be used. If the FRSTOR instruction were used, the operand size override prefix would be incorrectly applied to the implicit WAIT instruction which precedes FRSTOR, rather than to the save instruction itself.

Support for Power Managed Peripherals

SHADOW REGISTERS

Before power is removed from any device, the state of that device must be saved in a protected memory space so that the device can be reinitialized to its previous state. If a peripheral contains a write only register, the value in that register can be recovered by providing shadow registers that are both readable and writeable.

These shadow registers should be updated every time the peripheral registers are written, but they have no function other than tracking the data written to a particular register.

Example D-2. Saving the FPU State in 32-Bit Protected Mode

```
;first initialize the registers used to store the state save information
      mov   dx,SEGMENT          ;SEGMENT is the segment to be used by
                                ;the save instruction,
      mov   ds,dx               ; normally it should point to SMRAM
      mov   si,OFFSET           ;OFFSET is the offset used in the save
                                ;instruction

;set the PE bit in CR0
      mov   eax,cr0             ;read the old value of CR0
      or    eax,00000001H       ;set the PE bit
      mov   cr0, eax

;enter protected mode by executing a short jump to clear the prefetch queue
      jmp   protect
protect:

;we can now save the state of the FPU in the protected mode format

      db    66H                 ;use an operand size override prefix
                                ;to set 32-bit format
      fnsave[si]                ;FPU state saved to SEGMENT:OFFSET

;now return to real mode to continue with the SMM handler (no jump is
;required)

      mov   eax,cr0             ;clear the PE bit in CR0
      and   eax,0FFFFFFFEH
      mov   cr0,eax
```

271329-H2

In addition to the write only registers in a system, there are several other registers that must be shadowed. Any device that requires registers to be programmed in a particular sequence must also have its registers shadowed. Examples in a typical personal computer include the programmable interrupt controller, the DMA controller, and the programmable timer/counter.

It is also possible to perform shadowing of some write only registers using SMM. Any time a write cycle is generated to a write only register, the system can generate an SMI#. The SMM handler can use the processor state information saved in the SMM state save to save the data from the interrupted I/O cycle to a predetermined location in the SMRAM space.

Example D-3. Restoring the FPU State from a 32-Bit Protected Mode Save

```
;first initialize the registers used to recall the state save information

      mov   dx,SEGMENT          ;SEGMENT is the segment to be used by
                                ;the restore instruction,
      mov   ds,dx               ;normally it should point to SMRAM
      mov   si,OFFSET           ;OFFSET is the offset used in the
                                ;restore instruction

;set the PE bit in CR0

      mov   eax,cr0             ;read the old value of CR0
      or    eax,00000001H       ;set the PE bit
      mov   cr0, eax

;enter protected mode by executing a short jump to clear the prefetch queue

      jmp   protect
protect:

;we can now recall the state of the FPU from the previous FNSAVE instruction
;(in the protected mode format)

      db    66H                 ;use an operand size override prefix
                                ;to set 32-bit format
      fnrstor     [si]          ;FPU state restored from
                                ;SEGMENT:OFFSET

;now return to real mode to continue with the SMM handler (no jump is
;required)

      mov   eax,cr0      ;clear the PE bit in CR0
      and   eax,FFFFFFFEH
      mov   cr0,eax
```

271329–H3

The information contained in the SMM state save can be used (with the knowledge that the SMI# was in response to an I/O write instruction) to determine both the address and the data of the interrupted write instruction. The SMM handler can examine the OPCODEs of previous instructions by decrementing the IP (or EIP) register. Once the correct OPCODE is determined, it can be used with the values in the EAX and DX slots of the SMM state save to update the information in the memory used to shadow the I/O register. I/O write instructions occur in one of three forms: 1) a write to an address that is specified in the OPCODE; 2) a write to an address contained in the DX register; or 3) a string write to an address contained in the DX register.

The I/O write instructions have the following OPCODEs:

Table D-1. I/O Write Instruction OPCODEs

Instruction	OPCODE	Notes
OUT x,al	E6x	x is the address of the I/O port
OUT x,ax	E7x	x is the address of the I/O port
OUT x,eax	E7x	x is the address of the I/O port
OUT dx,al	EE	
OUT dx,ax	EF	
OUT dx,eax	EF	
OUTSB	6E	
OUTSW	6F	
OUTSD	6F	